Operating System: Persistency (Part 1) File System Implementation

Nguyen Tan Trung

Outline

- I/O devices
- Files and Directories
- File System Implementation

I/O devices

1. What is I/O devices

- Input Devices: An input device is any hardware component that allows you to enter data and instructions onto a computer. Six Widely used input devices are the keyboard, mouse, microphone, scanner, digital camera and PC video camera.
- Output Devices: An output device is any hardware component that conveys information to one or more people. Three commonly used output devices are a printer, a monitor and speakers.

I/O devices

2. System Architecture

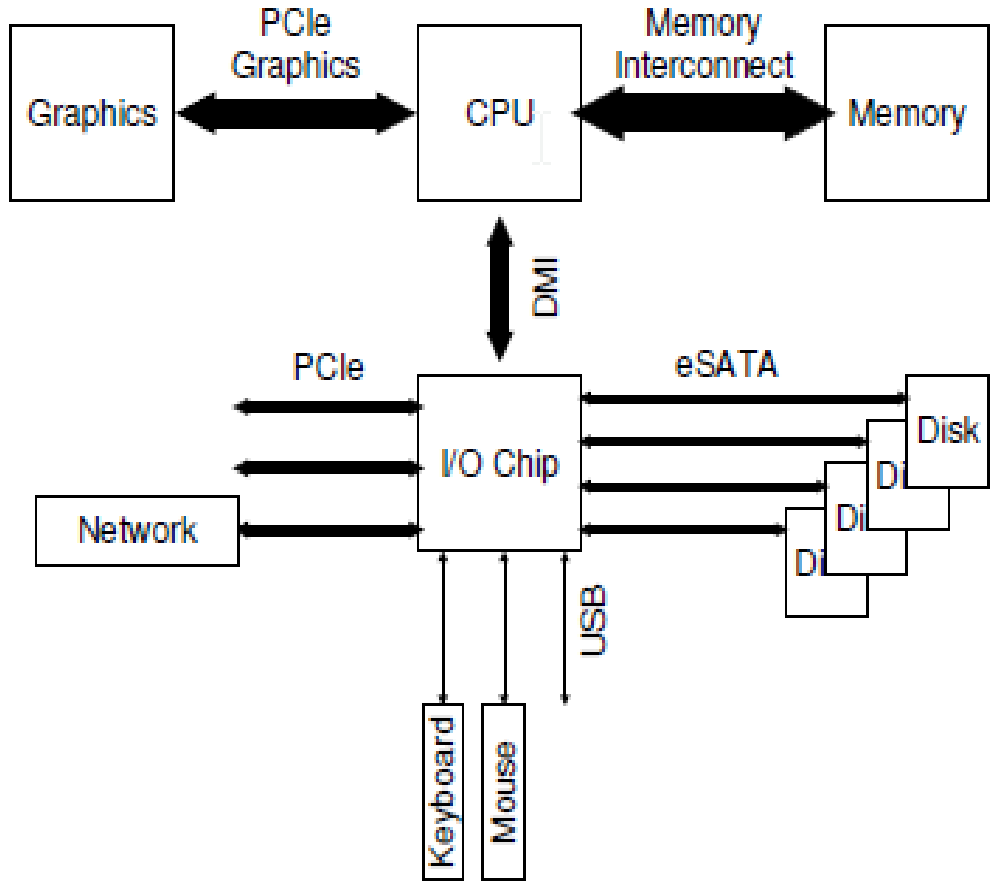


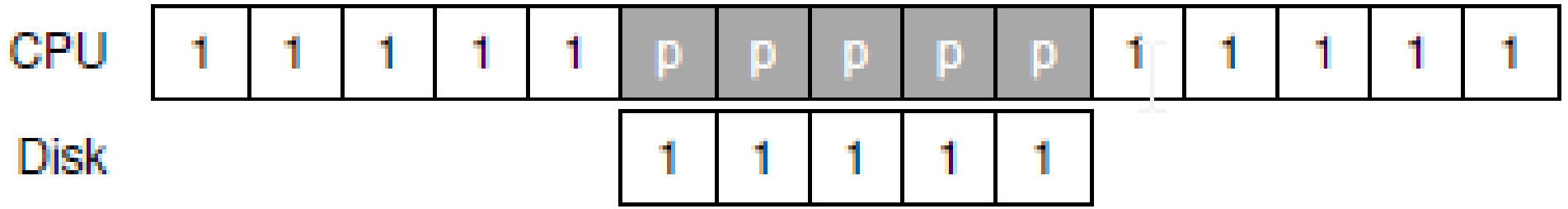
Figure 36.2: **Modern System Architecture**

- The CPU connects most closely to the memory system but also has a high-performance connection to the graphics card.
- The CPU connects to an I/O chip via **DMI** (Direct Media Interface).
- The rest of the devices connect to this chip via a number of different interconnects:
  - **eSATA**: external **SATA**, **ATA** (the AT Attachment), **SATA** (Serial ATA).
  - **PCIe:** Peripheral Component Interconnect Express
  - **USB:** Universal Serial Bus

I/O devices

3. Lowering CPU Overhead With Interrupts

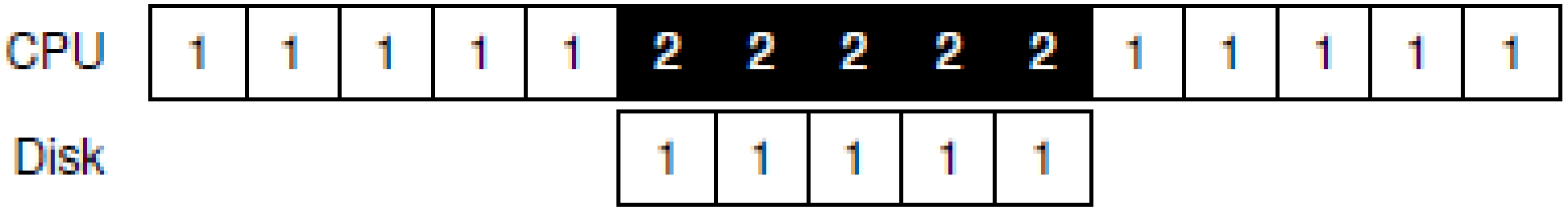
- **The problem**: In the diagram, Process 1 runs on the CPU for some time (indicated by a repeated 1 on the CPU line), and then issues an I/O request to the disk to read some data. Without interrupts, the system simply spins, polling the status of the device repeatedly until the I/O is complete (indicated by a p). The disk services the request and finally Process 1 can run again.



I/O devices

3. Lowering CPU Overhead With Interrupts

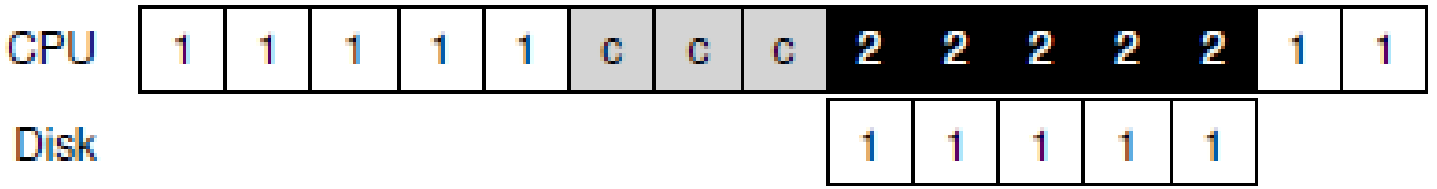
- **The solution**: We utilize interrupts and allow for overlap, the OS can do something else while waiting for the disk. the OS runs Process 2 on the CPU while the disk services Process 1's request. When the disk request is finished, an interrupt occurs, and the OS wakes up Process 1 and runs it again.



I/O devices

4. More Efficient Data Movement With DMA

- **The problem**: When using programmed I/O (PIO) to transfer a large chunk of data to a device, the CPU is once again overburdened with a rather trivial task, and thus wastes a lot of time and effort that could better be spent running other processes.



Process 1 is running and then wishes to write some data to the disk. It then initiates the I/O, which must copy the data from memory to the device explicitly, one word at a time (marked c in the diagram). When the copy is complete, the I/O begins on the disk and the CPU can finally be used for something else

I/O devices

4. More Efficient Data Movement With DMA

- **The solution**: We refer to as **Direct Memory Access (DMA)**. A DMA engine is essentially a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention

From the timeline, you can see that the copying of data is now handled by the DMA controller. Because the CPU is free during that time, the OS can do something else, here choosing to run Process 2. Process 2 thus gets to use more CPU before Process 1 runs again

I/O devices

5. Methods Of Device Interaction

- The first, oldest method (used by IBM mainframes for many years) is to have explicit **I/O instructions:** These instructions specify a way for the OS to send data to specific device registers and thus allow the construction of the protocols described above.
- For example, on x86, the **in** and **out** instructions can be used to communicate with devices. For example, to send data to a device, the caller specifies a register with the data in it, and a specific port which names the device.

```
IN   AL, 19H   ;8-bits are saved to AL from I/O port 19H.
IN   EAX, DX   ;32-bits are saved to EAX.
OUT  DX, EAX   ;32-bits are written to port DX from EAX.
OUT  19H,AX    ;16-bits are written to I/O port 0019H.
```

I/O devices

5. Methods Of Device Interaction

- The second method to interact with devices is known as **memory-mapped I/O:** The hardware makes device registers available as if they were memory locations. To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

A sample system memory map

Address range (hexadecimal)	Size	Device
0000–7FFF	32 KiB	RAM
8000–80FF	256 bytes	General-purpose I/O
9000–90FF	256 bytes	Sound controller
A000–A7FF	2 KiB	Video controller/text-mapped display RAM
C000–FFFF	16 KiB	ROM

For I/O devices

I/O devices

6. Fitting Into The OS: The Device Driver

- At the lowest level, a piece of software in the OS must know in detail how a device works. We call this piece of software a **device driver**, and any specifics of device interaction are encapsulated within.

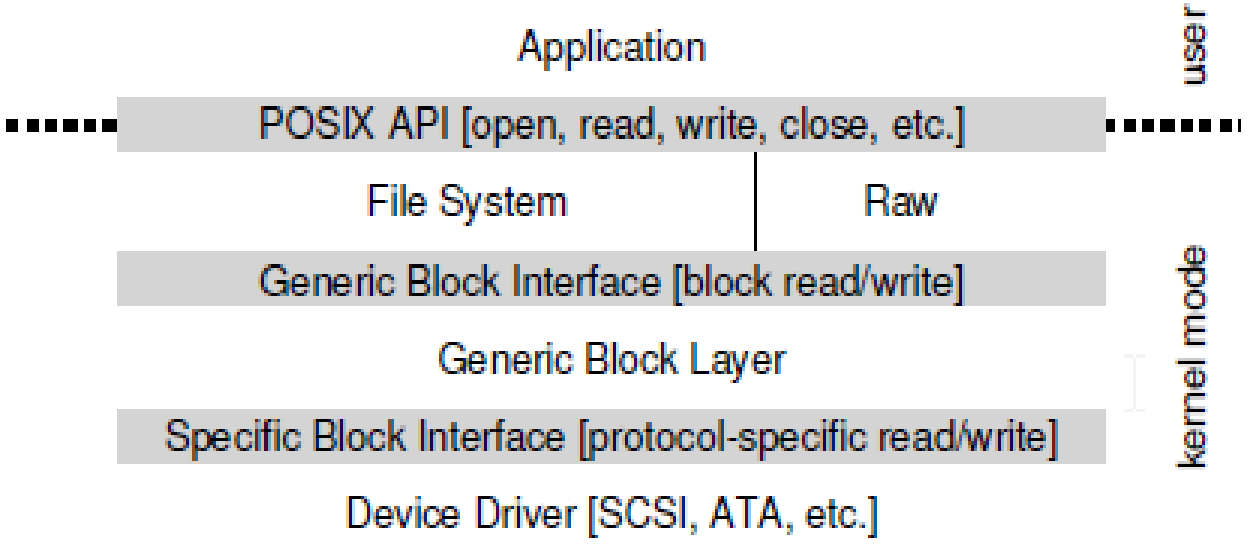


Figure 36.4: **The File System Stack**

- The file system simply issues block read and write requests to the generic block layer.
- The generic block layer routes them to the appropriate device driver.
- The device driver handles the details of issuing the specific request

I/O devices

7. Case Study: A Simple IDE Disk Driver

```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset,
                  E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6     5      4     3     2     1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when ERROR==1)
    7      6     5      4     3     2     1     0
   BBK    UNC   MC    IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

Figure 36.5: **The IDE Interface**

I/O devices

7. Case Study: A Simple IDE Disk Driver

```
1   static int ide_wait_ready() {
2       while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
3       ; // loop until drive isn't busy
4   }
5   static void ide_start_request(struct buf *b) {
6       ide_wait_ready();
7       outb(0x3f6, 0); // generate interrupt
8       outb(0x1f2, 1); // how many sectors?
9       outb(0x1f3, b->sector & 0xff); // LBA goes here ...
10      outb(0x1f4, (b->sector >> 8) & 0xff); // ... and here
11      outb(0x1f5, (b->sector >> 16) & 0xff); // ... and here!
12      outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
13      if(b->flags & B_DIRTY){
14          outb(0x1f7, IDE_CMD_WRITE); // this is a WRITE
15          outsl(0x1f0, b->data, 512/4); // transfer data too!
16      } else {
17          outb(0x1f7, IDE_CMD_READ); // this is a READ (no data)
18      }
19  }
```

I/O devices

7. Case Study: A Simple IDE Disk Driver

```
20  void ide_rw(struct buf *b) {
21      acquire(&ide_lock);
22      for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
23      ; // walk queue
24      *pp = b; // add request to end
25      if (ide_queue == b) // if q is empty
26          ide_start_request(b); // send req to disk
27      while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
28          sleep(b, &ide_lock); // wait for completion
29      release(&ide_lock);
30  }
31  void ide_intr() {
32      struct buf *b;
33      acquire(&ide_lock);
34      if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
35          insl(0x1f0, b->data, 512/4); // if READ: get data
36      b->flags |= B_VALID;
37      b->flags &= ~B_DIRTY;
38      wakeup(b); // wake waiting process
39      if ((ide_queue = b->qnext) != 0) // start next request
40          ide_start_request(ide_queue); // (if one exists)
41      release(&ide_lock);
42  }
```

Files and Directories

1. What are Files and Directories

- A persistent-storage device, such as a classic hard disk drive or a more modern solid-state storage device, stores information permanently (or at least, for a long time).
- Two key abstractions have developed over time in the virtualization of storage: Files and Directories
- A file is simply a linear array of bytes, each of which you can read or write. Each file has some kind of **low-level name** that is often referred to as its **inode number.**
- A directory, like a file, also has a low-level name (i.e., an inode number), but its contents are quite specific: it contains a list of (user-readable name, low-level name) pairs.

Files and Directories

2. Directories hierarchy

By placing directories within other directories, users are able to build an arbitrary **directory tree** (or **directory hierarchy**), under which all files and directories are stored.

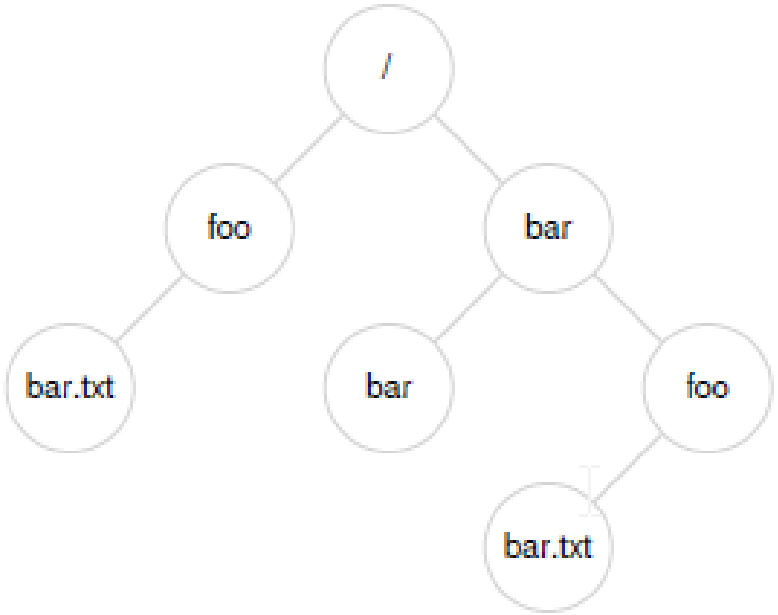


Figure 39.1: **An Example Directory Tree**

- A directory tree or directory hierarchy organizes all files and directories into a large tree, starting at the root

Files and Directories

3. The File System Interface

There is an overview of the APIs: the interfaces you'll expect to see when interacting with a UNIX file system

- Creating Files:
- Reading And Writing Files
- Reading And Writing, But Not Sequentially
- Shared File Table Entries
- Writing Immediately With ***fsync()***
- Renaming Files
- Getting Information About Files
- Removing Files
- Making Directories
- Reading Directories
- Deleting Directories

Files and Directories

3. The File System Interface

- Creating Files: This can be accomplished with the open system call; by calling **open()** and passing it the **O_CREAT** flag, a program can create a new file.

Example: to create a file called “foo” in the current working directory

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
S_IRUSR|S_IWUSR);
```

Files and Directories

3. The File System Interface

- Reading And Writing Files : Using the **read()** system call to read some bytes from a file. Using the **write()** system call to write some bytes to a file.

```
int fd = open("file", O RDONLY);
read(fd, buffer, 100);
read(fd, buffer, 100);
read(fd, buffer, 100);
close(fd);
```

```
int fd = open("file", O_WRONLY);
int rc = write(fd, buffer, size);
assert(rc == size);
close(fd);
```

Files and Directories

3. The File System Interface

- Reading And Writing, But Not Sequentially : Using the **read()/write()** and **lseek()** system call.

```
int fd = open("file", O RDONLY);
lseek(fd, 200, SEEK SET);
read(fd, buffer, 100);
close(fd);
```

Files and Directories

3. The File System Interface

- Shared File Table Entries : There are a few interesting cases where an entry in the open file table is shared. One of those cases occurs when a parent process creates a child process with ***fork()/*dup()**.

```
prompt> ./fork-seek
child: offset 10
parent: offset 10
prompt>
```

File : fork-seek.c

```
int main(int argc, char *argv[]) {
        int fd = open("file.txt", O_RDONLY);
        assert(fd >= 0);
        int rc = fork();
        if (rc == 0) {
                rc = lseek(fd, 10, SEEK_SET);
                printf("child: offset %d\n", rc);
        } else if (rc > 0) {
                (void) wait(NULL);
                printf("parent: offset %d\n",
                (int) lseek(fd, 0, SEEK_CUR));
        }
        return 0;
}
```

Files and Directories

3. The File System Interface

- Shared File Table Entries :

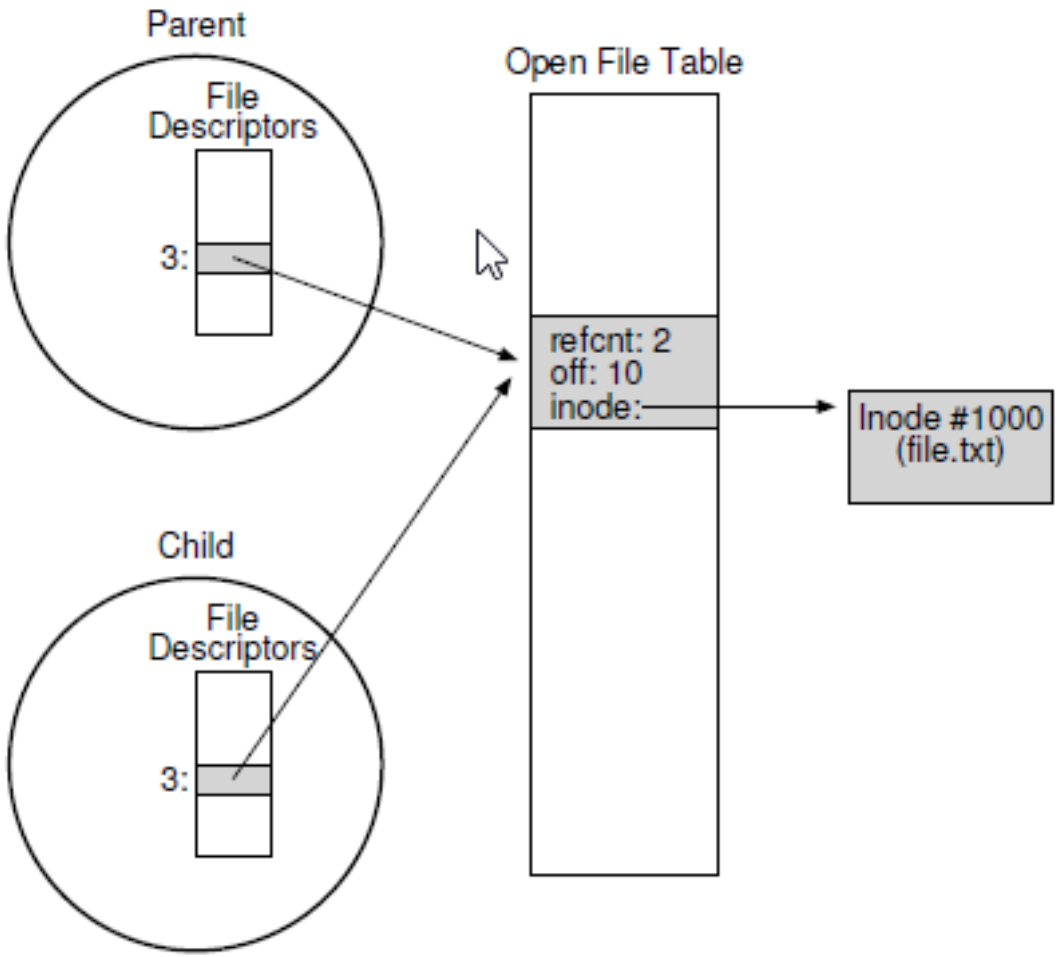


Figure 39.3: **Processes Sharing An Open File Table Entry**

Files and Directories

3. The File System Interface

- Writing Immediately With ***fsync()*** : The ability to force writes to disk from time to time, help to avoid the machine crashes after the write() call.

```
int fd = open("foo",
        O_CREAT|O_WRONLY|O_TRUNC,
        S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

Files and Directories

3. The File System Interface

- Renaming Files : Using the system call ***rename(char *old, char *new)***, which takes precisely two arguments: the original name of the file (old) and the new name (new).

```
int fd = open("foo.txt.tmp",
        O_WRONLY|O_CREAT|O_TRUNC,
        S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Files and Directories

3. The File System Interface

- Getting Information About Files: Beyond file access, we expect the file system to keep a fair amount of information about each file it is storing. We generally call such data about files **metadata**. To see the metadata for a certain file, we can use the **stat()** or **fstat()** system calls.

```
struct stat stats;
// stat() returns 0 on successful operation,
// otherwise returns -1 if unable to get file properties.
if (stat(path, &stats) == 0){
        printFileProperties(stats);
} else {
        printf("Unable to get file properties.\n");
        printf("Please check whether '%s' file exists.\n", path);
}
```

```
struct stat {
        dev_t st_dev; // ID of device containing file
        ino_t st_ino; // inode number
        mode_t st_mode; // protection
        nlink_t st_nlink; // number of hard links
        uid_t st_uid; // user ID of owner
        gid_t st_gid; // group ID of owner
        dev_t st_rdev; // device ID (if special file)
        off_t st_size; // total size, in bytes
        blksize_t st_blksize; // blocksize for filesystem I/O
        blkcnt_t st_blocks; // number of blocks allocated
        time_t st_atime; // time of last access
        time_t st_mtime; // time of last modification
        time_t st_ctime; // time of last status change
};
```

Files and Directories

3. The File System Interface

- Removing Files: To remove a certain file, we can use the **unlink()** system calls.

```
char *path = "/modules/pass1";
int   status;
…
status = unlink(path);
```

Files and Directories

3. The File System Interface

- Making Directories: To create a directory, a single system call, **mkdir**(), is available.

```
int   status;

…

status = mkdir("/home/cnd/mod1",
S_IRWXU | S_IRWXG | S_IROTH |
S_IXOTH);
```

Files and Directories

3. The File System Interface

- Reading Directories: To read the contents of a directory, uses three calls, **opendir**(), **readdir**(), and **closedir**()

```
int main(int argc, char *argv[]) {
        DIR *dp = opendir(".");
        assert(dp != NULL);
        struct dirent *d;
        while ((d = readdir(dp)) != NULL) {
                printf("%lu %s\n", (unsigned long) d->d_ino, d->d_name);
        }
        closedir(dp);
        return 0;
}
```

```
struct dirent {
char d_name[256]; // filename
ino_t d_ino; // inode number
off_t d_off; // offset to the next dirent
unsigned short d_reclen; // length of this record
unsigned char d_type; // type of file
};
```

Files and Directories

3. The File System Interface

- Deleting Directories: To delete a directory, a single system call, **rmdir**(), is available.

```
int   status;
…
status =
rmdir("/home/cnd/mod1");
```

File System Implementation

1. What is File System Implementation

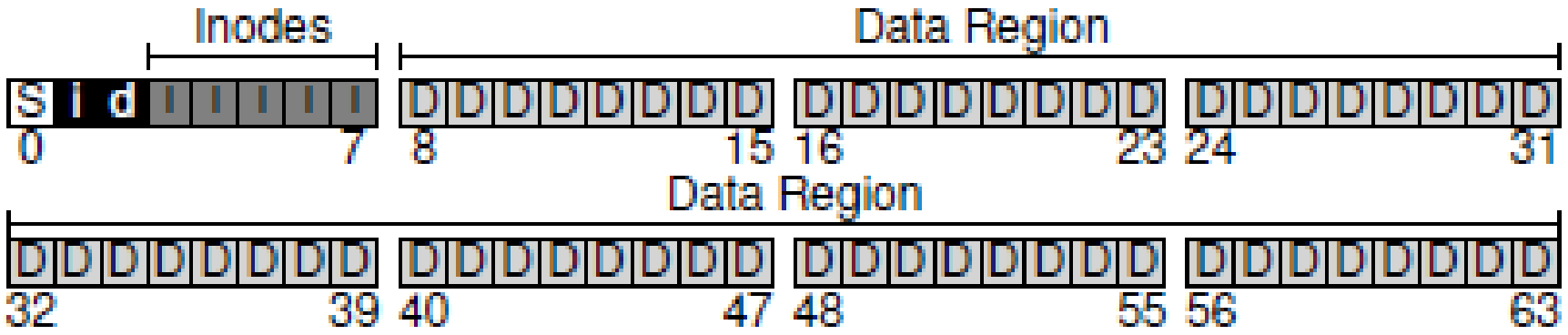
File system implementation defines:

- How files and directories are stored.
- How disk space is managed.
- How to make everything work efficiently and reliably.

File System Implementation

2. Overall Organization

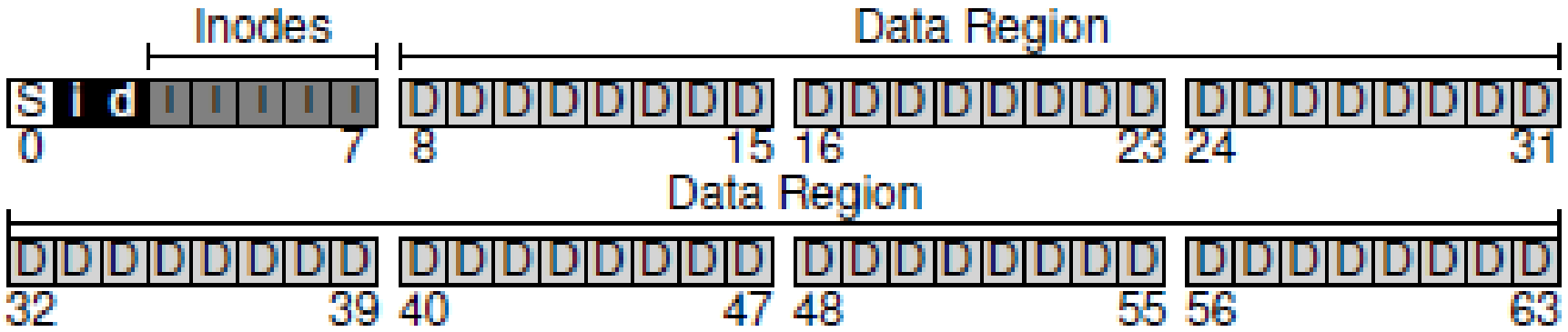
- Superblock(S): contains information about this particular file system, including, for example, how many inodes and data blocks are in the file system (80 and 56, respectively in this instance), where the inode table begins (block 3), and so forth, a magic number of some kind to identify the file system type (in this case, vsfs).
- inode bitmap (i)/ data bitmap (d): each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1).



File System Implementation

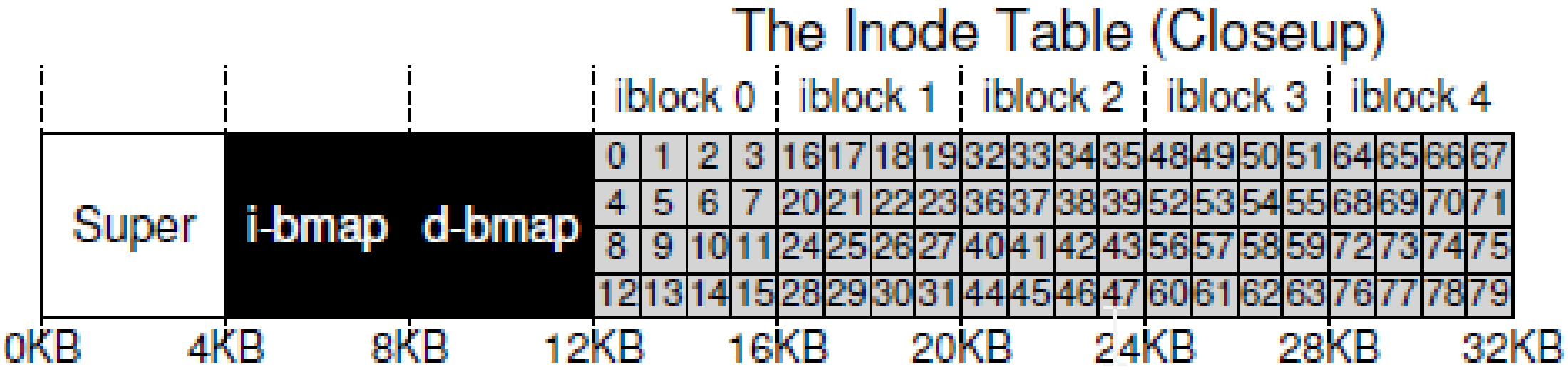
2. Overall Organization

- inodes(I): a inode use to track information about each file like which data blocks (in the data region) comprise a file, the size of the file, its owner and access rights, access and modify times, and other similar kinds of information.
- data region(D): data stored in each file.



File System Implementation

3. File Organization: The Inode



File System Implementation

3. File Organization: The Inode

- The name inode is short for **index node**. Nodes were originally arranged in an array, and the array ***indexed*** into when accessing a particular inode.
- Each inode is implicitly referred to by a number (called the **i-number**), which we've earlier called the **low-level name** of the file.
- The sector address ***sector*** of the inode block can be calculated as follows:

  sector = ((inumber * sizeof(inode_t)) *+ inodeStartAddr) / sectorSize*

  Example: inumber: 32, sizeof(inode_t): 256 bytes, inodeStartAddr = 12KB,
  sectorSize = 512 bytes => sector = ((32 * 256) *+* 12*1024*) / 512 = 40*

File System Implementation

3. File Organization: The Inode

- Inside each inode is virtually all of the information need about a file.

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figure 40.1: **Simplified Ext2 Inode**

File System Implementation

4. Directory Organization

- A directory basically just contains a list of (entry name, inode number) pairs.
- For example, assume a directory ***dir*** (inode number 5) has three files in it (***foo***, ***bar***, and ***foobar is a pretty longname***), with inode numbers 12, 13, and 24 respectively. The on-disk data for dir might look like:

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

File System Implementation

4. Free Space Management

- A file system must track which inodes and data blocks are free, and which are not, so that when a new file or directory is allocated, it can find space for it.
- We have two simple bitmaps(for inode and data) for this task.

File System Implementation

5. Access Paths: Reading and Writing

- Reading A File From Disk

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			read			read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

File System Implementation

5. Access Paths: Reading and Writing

- Writing A File To Disk

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read	read write write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

File System Implementation

6. Caching and Buffering

- Most file systems aggressively use system memory (DRAM) to cache important blocks.
- Reading and writing files can be expensive, incurring many I/Os to the (slow) disk.
- With caching, the first open may generate a lot of I/O traffic to read in directory inode and data, but sub-sequent file opens of that same file (or files in the same directory) will mostly hit in the cache and thus no I/O is needed.
- With write buffering, some writes are avoided altogether by delaying them.

Conclusion

- Basic understanding of how an OS interacts with a device.
- Two techniques, the interrupt and DMA, have been introduced to help with device efficiency.
- Two approaches to accessing device registers, explicit I/O instructions and memory-mapped I/O, have been described
- The notion of a device driver has been presented.
- Two key abstractions have developed over time in the virtualization of storage.
- Discuss the file system interface in more detail.
- Basic machinery required in building a file system.

Reference

- Operating Systems: Three Easy Pieces (wisc.edu)

THANK YOU